实验五 流水线 MIPS 处理器设计

无 73
2016011866
欧阳良扬
2019 年 9 月 4 日

一、 实验目的

1. 将实验四设计的单周期 MIPS 处理器改进为流水线结构；
2. 增加异常与中断的处理机制；
3. 增加定时器、数码管等外设；

二、 设计方案

1. 指令集

本次实验实现了以下几种类型的 MIPS 指令：

空指令 nop

R 型算术/逻辑指令 add addu sub subu and or xor nor sll srl sra slt

I 型算术/逻辑指令 addi addiu andi ori slti sltiu lui

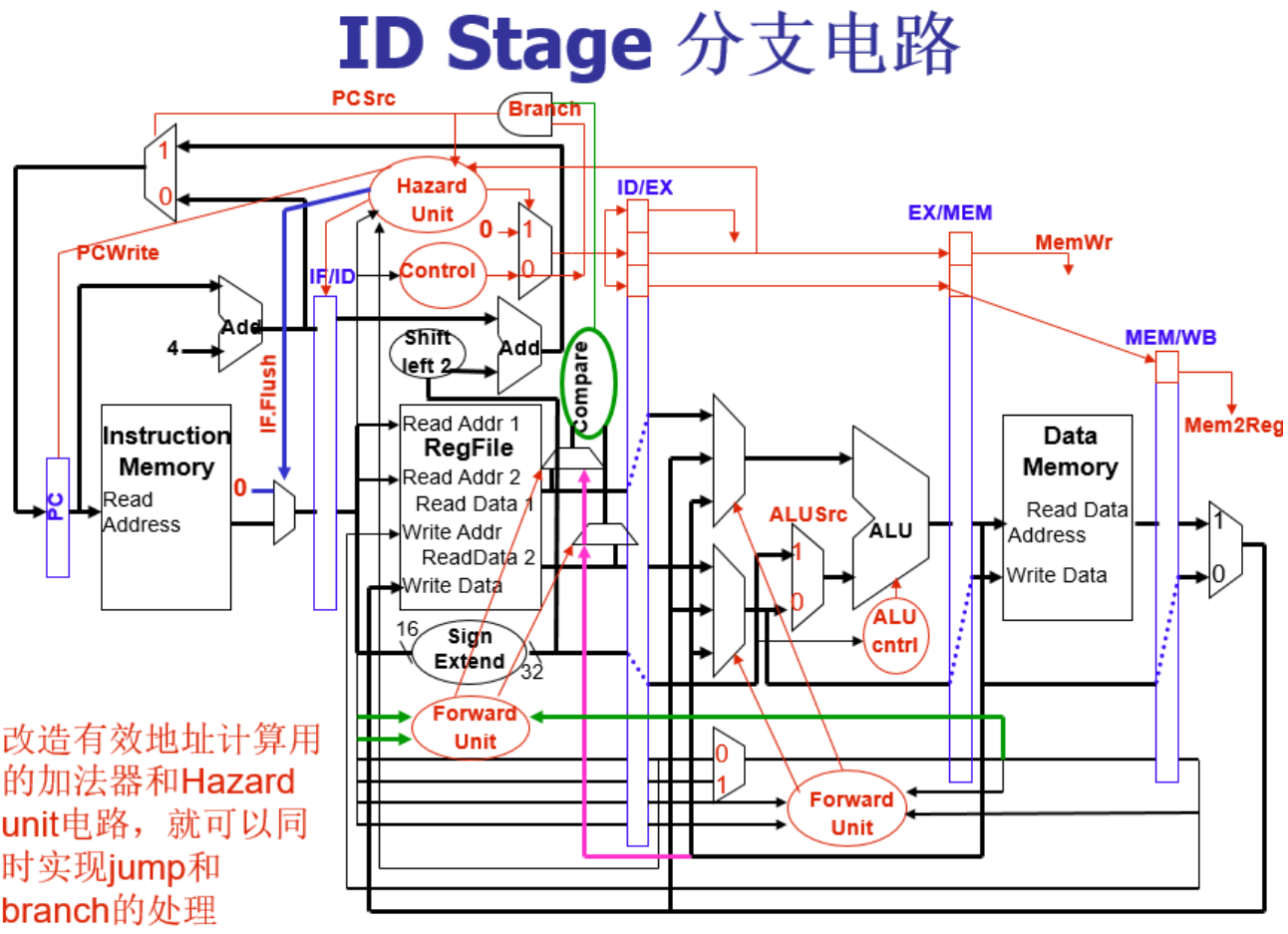
储存访问指令 lw sw

分支与跳转指令 beq bne bltz bgtz j jal jr jalr

我们的设计过程也是按照上述顺序，先设计只能处理算术型指令的流水线结构处理器，再增添储存、分支、跳转乃至异常与中断处理功能，循序渐进地完成本次实验设计。

2. 流水线处理器结构

我们的处理器结构基本是按照理论课 PPT 第 13 讲 134 页的图示进行设计的：



比较该结构与单周期处理器的结构，有许多部分是可以直接使用的，如 ALU,RegFile,DataMemory 和大部分的 Control,在本报告中我们就不再赘述。然后按照五级流水线的设计思想，将控制信号、数据等划分为 IF,ID,EX,MEM 和 WB 五个部分。这五个部分本身是逻辑电路，他们接受上图中蓝色的级间寄存器的数据和控制信号，产生对应的输出数据，并在下一个时钟上升沿将数据和控制信号送到下一级。下面我们将介绍每一级流水线的输入数据、控制信号以及输出（data 表示数据，sig 表示控制信号，变量名后面的大写字母表示该数据或控制信号来自哪一级流水线）。

2.1 IF 级

变量名	长度	类型	说明
clk,reset	1	input sig	系统时钟
stall,flush	1	input sig	流水线数据/控制冒险产生的阻塞 当产生 stall 时，上一条指令将被移动到本条的位置 当产生 flush 时，本条指令应当是空指令
PCSrcID	3	input sig	在 ID 阶段产生，决定下一条 PC(PCnext) 3'd0:PC+4　　3'd1:branchaddrID　　3'd2:jumpaddrID 3'd3:interrupt, 0x80000004　　3'd4:exception,0x80000008
branchaddrID jumpaddrID instructionID	32	input data	ID 阶段产生的分支地址 ID 阶段的 instruction 被用于 stall 信号
instructionIF PCIF PCplus4IF	32	output data	IF 级的输出 产生 stall 时，PCIF 不变，PCplus4IF 和 PCnext 保持为 PC instructionIF 保持为 instructionID 产生 flush 时，PCIF 变为 PC-4，instructionIF 置零

2.2 IF/ID 级间寄存器

变量名	长度	类型	说明
clk,reset	1	input sig	系统时钟
PCIF PCplus4IF instructionIF	32	input data	IF 级的输出
PCID PCplus4ID instructionID	32	output data	ID 级的输入

2.3 ID 级

变量名	长度	类型	说明
clk,reset	1	input sig	系统时钟
stall	1	input sig	流水线数据/控制冒险产生的阻塞 当产生 stall 时，强制令一些控制信号为 0
exception interrupt	1	sig input sig	未定义指令产生的异常 来自外设的中断 当产生异常/中断时，强制令一些控制信号为 0
regwriteaddrWB regwriteaddrMEM	5	input data	MEM 和 WB 阶段的指令将要写入的寄存器地址 用于完成 MEM,WB 到 ID 阶段的转发
regwritedataWB ALUoutMEM	32	input data	MEM 和 WB 阶段的指令将要写入的数据 用于完成 MEM,WB 到 ID 阶段的转发
RegWriteWB RegWriteMEM	1	input sig	MEM 和 WB 阶段的指令是否需要写回的控制信号
readdata1ID readdata2ID extenddataID branchaddrID jumpaddrID	32	output data	从寄存器中读出的数据（经过转发）/立即数扩展后的数据 跳转地址亦在此时被计算出来
rd/t/saddrID,shamtID	5	output data	从指令中读出的寄存器地址信息

RegWriteID ExtOpID MemReadID MemWriteID ALUSrcID BranchID JumpID JRID	1	output sig	是否有寄存器写回 立即数扩展逻辑 是否从储存中读数 是否向储存中写数 ALU 输入来自寄存器或是立即数扩展 是否是 beq,bne,bgtz,bltz 指令 是否是 j,jr,jal,jalr 指令 是否是 jr,jalr 指令
RegDstID MemtoRegID	2	output sig	决定写回的寄存器 2'd0:rt, 2'd1:rd, 2'd2:$31, 2'd3:$26 决定写回的数据 2'd0:ALUout, 2'd1:memreaddata 2'd2:PCplus4 2'd4"PC
PCSrcID	3	output sig	见 IF 阶段
ALUop	4	output sig	ALU 的操作
FunctID	6	output sig	ALU 的操作

2.4 ID/EX 级间寄存器

上述 ID 级除了 ExtOp,JRID,JumpID,BranchID 这些 ID 阶段就使用完毕的信号，其他输出数据与控制信号均通过级间寄存器送到 EX 级。另外，PCID 与 PCplus4ID 也直接从 IF/ID 级间寄存器送到 ID/EX 级间寄存器。

2.5 EX 级

变量名	长度	类型	说明
clk,reset	1	input sig	系统时钟
readdata1EX readdata2EX extenddataEX	32	input data	从寄存器中读出的数据/立即数扩展后的数据
regwritedataWB ALUoutMEM	32	input data	MEM 和 WB 阶段的指令将要写入的数据 用于完成 MEM,WB 到 EX 阶段的转发
regwriteaddrWB regwriteaddrMEM	5	input data	MEM 和 WB 阶段的指令将要写入的寄存器地址 用于完成 MEM,WB 到 EX 阶段的转发
RegWriteWB RegWriteMEM	1	input sig	MEM 和 WB 阶段的指令是否需要写回的控制信号
rd/t/saddrEX	5	input data	从指令中读出的寄存器地址信息 用于判断出 regwriteaddrEX 和转发单元
ALUSrcEX	1	input sig	决定 ALUin2 是立即数还是寄存器读出的数
RegDstEX	2	input sig	见 ID 阶段
ALUOpEX FunctEX	4 6	input sig	ALU 操作
ALUoutEX memwritedataEX	32	output data	ALU 输出 sw 使用的数据
regwriteaddrEX	5	output data	写回寄存器地址

2.6 EX/MEM 级间寄存器

EX 级的输出将送到 MEM 级。在 EX 级没有用到的 MemWriteEX,MemReadEX,MemtoRegEX,RegWriteEX,PCEX 和 PCplus4EX 也都直接送到下一级。

2.7 MEM 级

我们先讨论无外设的 MEM 级，外设部分将在下面单独说明

变量名	长度	类型	说明
clk,reset	1	input sig	系统时钟
memaddr memwritedata	32	input data	内存地址，实际就是 ALUoutMEM 写入数据，注意其也在 EX 经过转发
MemReadMEM MemWriteMEM	1	input sig	写入与读出
MemreaddataMEM	32	output data	读出的数据

2.8 MEM/WB 级间寄存器

将 ALUoutMEM,memreaddataMEM,PCMEM,PCplus4MEM 这四个可能被写回的数据及地址 regwriteaddrMEM，加上控制信号 RegWriteMEM，MemtoRegMEM 传递到 WB 级。

2.9 WB 级

WB 级的部分功能已经在 ID 阶段说明。

变量名	长度	类型	说明
clk,reset	1	input sig	系统时钟
ALUoutWB memreaddataWB PCWB PCplus4WB	32	input data	可能被写回的数据
MemtoRegWB	2	input sig	数据选择
regwritedataWB	32	output data	被写回的数据

2.10 转发模块

由于我们采取了 beq 在 ID 阶段产生跳转结果，因此一共涉及四种可能的数据转发：

WB_ID	这一转发实际上就是所谓寄存器堆的“先写后读”，若 ID 阶段取数的寄存器与 WB 阶段写回的寄存器相同，则进行转发。我们可以采用下降沿触发的方式减少这一转发操作，但 FPGA 往往对上升沿有优化，因此这里我采用转发实现“先写后读”。
WB_EX	若 EX 阶段取数的寄存器与 WB 阶段写回的寄存器相同则进行转发。这一转发存在于 load-use 型阻塞，或是相差一条指令的情况中。
MEM_ID	若 ID 阶段取数的寄存器与 MEM 阶段写回的寄存器相同则进行转发。注意这一转发只适用于写回数据是 ALUoutMEM 的情况，若写回数据是 PC 或是内存中的值，则需要阻塞一个周期
MEM_EX	若 EX 阶段取数的寄存器与 MEM 阶段写回的寄存器相同则进行转发。同样只适用于写回数据是 ALUoutMEM 的情况。

另外当两个转发发生冲突时，应该转发更近的那一条，即 MEM 优先于 WB。

2.11 阻塞

有两种不同的阻塞信号，分别是 stall 和 flush。stall 是数据冒险导致本指令使用的操作数不得不阻塞一个周期以获得正确的值，而 flush 是控制冒险导致一些执行到一半的指令被清空。由于我们把跳转都放到 ID 阶段，stall 和 flush 也就只在 IF 和 ID 阶段发挥作用，可见上述表格。这里我们讨论他们的产生逻辑。

我们的处理器结构的 PCSrc 是在 ID 阶段产生的（包括中断、异常也都是打断处于 ID 阶段的这条指令），因此 flush 逻辑非常简单，即(stall!=1 & PCSrcID!=0)。而 stall 的情况比较复杂，在我们的设计中共有 9 种产生 stall 的逻辑：（下表中第一行$表示写回的寄存器，r 表示算术/逻辑指令，第三行$表示要读的寄存器）

lw $1	r $1	lw $1	jal $31	中断/异常	r $1	lw $1	jal $31	中断/异常
stall	stall	stall*2	flush stall	flush stall	stall	stall*2	flush stall	flush stall
r $1	beq	beq $1	beq $31	beq $26	jr $1	jr $1	jr $31	jr $26

总结来说，由于 beq 和 jr 都是在 ID 阶段就要获取最新的寄存器值，因此不得不阻塞。同时，lw,jal 和异常中断都要到 WB 阶段才能获取正确的值，因此当他们与 beq,jr 同时出现时要阻塞两个周期。但由于 jal 和异常中断涉及一个 flush 周期，因此也只需要一个 stall。jal 指令后接的 r 指令如果用到$31 理论上也要一个 stall，但由于 jal 自带一个 flush，这个 stall 也就省去了。当然，我们可以加入新的转发模块使得 PC,PCplus4 也被及时转发，但由于这些情况实际上非常难出现，我们就通过 stall 来解决这些问题了。

3. 异常与中断

首先介绍异常处理。我们的处理器支持未定义指令异常，在 ID 阶段检测到未定义指令时，PCSrcID 置 4，同时控制信号将写回寄存器设置为$26，写回数据设置为 PCplus4，MemWrite 置零。如此在异常处理程序结束后，通过 jr $26 这条指令就可以回到发生异常的指令的下一条继续工作。

然后介绍中断。中断信号由外设中的定时器产生，当其为 1 时，PCSrcID 置 3，同时控制信号将写回寄存器设置为$26，写回数据设置为 PC，MemWrite 置零。在中断处理程序结束后，通过 jr $26 回到发生中断的指令继续工作。在本次实验中，我们用中断程序来设定数码管的状态，通过定时器不断发起中断，实现数码管四位数字的显示。

为了防止在中断、异常中再次发生中断、异常，PC 最高位被设置为监督位。在 IF 模块的介绍中可以看见，中断与异常的首地址是 0x80000004/8，这就实现了监督位置 1。而被存进$26 的 PC 值监督位是 0，以保证下次中断正常开展。

4. 外设

外设实际上可以看做存储器的一部分，只不过外设除了存储外还有额外功能。我们令外设的首地址为 0x40000000。外设的输入与输出见下表：

变量名	长度	类型	说明
clk,reset	1	input sig	系统时钟
memaddr memwritedata	32	input data	写入地址与数据
MemReadMEM MemWriteMEM	1	input sig	写入与读出
check	1	input sig	PC 监督位
interrupt	1	output sig	产生中断信号
preaddata	32	output data	读出的数据
digi,leds	12,8	output data	连接到外部数码管，led 等设备

按照实验指导上的代码，很容易就能实现系统时钟计数器、中断定时器和 led 等外设。需要注意的就是 interrupt 信号使用的监督位是 PCIF[31]而非 PCMEM，这样就保证了 interrupt 信号只在定时器计数到 0xffffffff 的那一个周期中产生，不对之后指令造成影响。

三、 关键代码

由于在编写代码时我是先写上层的 CPU.v 再往下逐个实现模块的，因此在各个文件中都有一些多余的端口。必要的端口在设计方案中都已经介绍，因此在以下关键代码中请忽略一些多余变量。

首先看顶层的 CPU.v

```
module CPU(reset, clk, leds, digi, showaddr, ramshowdata);

    input reset, clk;//reset,and clock
    input [6:0] showaddr;
   wire stall,flush;//stall for load-use,flush for beq,j
   wire intterupt,exception;
   output [7:0] leds;
   output [11:0] digi;
   output [31:0] ramshowdata;

   //wire PCSrcMEM;//control signal to decide the next PC
   wire [2:0] PCSrcID;//control signal to decide the next PC
   wire [31:0] jumpaddrID;//address of j,jal
   wire [31:0] branchaddrID;//address of branch
   //wire [31:0] regaddr;//address of $ra
```

```
    wire [31:0] instructionIF,instructionID;//instruction in IF,ID
    wire [31:0] PCplus4IF,PCIF;//PC and PC+4 in IF

    IF
IF(.clk(clk),.reset(reset),.intterupt(intterupt),.stall(stall),.flush(flush),.exception
(exception),
        .PCSrcID(PCSrcID),//input control signals
        .branchaddrID(branchaddrID),.instructionID(instructionID),.jumpaddrID(jumpaddrID
),//input data
        .instructionIF(instructionIF),.PCplus4IF(PCplus4IF),.PCIF(PCIF));//output data

    //wire [31:0] instructionID;//used in IF stall
    wire [31:0] PCplus4ID,PCID;

    IF ID IF ID(.clk(clk),.reset(reset),.intterupt(intterupt),
        .PCIF(PCIF),.PCplus4IF(PCplus4IF),.instructionIF(instructionIF),//input
        .PCID(PCID),.PCplus4ID(PCplus4ID),.instructionID(instructionID));//output

    wire [4:0] regwriteaddrWB,regwriteaddrMEM;//regwrite address from WB,MEM
    wire [31:0] regwritedataWB,ALUoutMEM;//regwrite data from WB,MEM
    wire RegWriteWB,RegWriteMEM;//regwrite control signal from WB,MEM
    wire [31:0] readdata1ID;
    wire [31:0] readdata2ID;//read data from registers
    wire [31:0] extenddataID;//imm extend data
    //wire [31:0] branchaddrID;
    wire [4:0] rdaddrID,rsaddrID,rtaddrID;//register dst address
    wire [4:0] shamtID;
    wire RegWriteID,ExtOpID,MemReadID,MemWriteID,ALUSrcID,BranchID,JumpID,JRID;
    wire [1:0] RegDstID,MemtoRegID;
    wire [3:0] ALUOpID;
    wire [5:0] FunctID;//control signals

    ID
ID(.clk(clk),.reset(reset),.intterupt(intterupt),.stall(stall),.exception(exception),
        .instructionID(instructionID),.PCplus4ID(PCplus4ID),.PCID(PCID),
        .regwriteaddrWB(regwriteaddrWB),.regwritedataWB(regwritedataWB),.RegWriteWB(RegW
riteWB),
        .regwriteaddrMEM(regwriteaddrMEM),.ALUoutMEM(ALUoutMEM),.RegWriteMEM(RegWriteMEM
),//input data and control signals
        .readdata1ID(readdata1ID),.readdata2ID(readdata2ID),.extenddataID(extenddataID),
        .rdaddrID(rdaddrID),.rtaddrID(rtaddrID),.rsaddrID(rsaddrID),.branchaddrID(branch
addrID),.jumpaddrID(jumpaddrID),//output data
        .RegWriteID(RegWriteID),.ExtOpID(ExtOpID),.MemReadID(MemReadID),.PCSrcID(PCSrcID
),.shamtID(shamtID),
        .MemWriteID(MemWriteID),.ALUSrcID(ALUSrcID),.MemtoRegID(MemtoRegID),.JumpID(Jump
ID),.JRID(JRID),
        .BranchID(BranchID),.RegDstID(RegDstID),.ALUOpID(ALUOpID),.FunctID(FunctID));//o
utput control signals

    wire [31:0] PCplus4EX,PCEX;
    wire [31:0] readdata1EX;
    wire [31:0] readdata2EX;//read data from registers
    wire [31:0] extenddataEX;//imm extend data
    wire [4:0] rdaddrEX,rtaddrEX,rsaddrEX;//register dst address
    wire [4:0] shamtEX;
    wire RegWriteEX,ExtOpEX,MemReadEX,MemWriteEX,ALUSrcEX;
    wire [1:0] RegDstEX,MemtoRegEX;
    wire [3:0] ALUOpEX;
    wire [5:0] FunctEX;//control signals

    wire [4:0] regwriteaddrEX;
    wire MemReadMEM;
    wire [1:0] MemtoRegMEM;
    StallFlush StallFlush(.clk(clk),.reset(reset),.intterupt(intterupt),
        .regwriteaddrEX(regwriteaddrEX),.BranchID(BranchID),.RegWriteEX(RegWriteEX),.JRI
D(JRID),
        .MemReadEX(MemReadEX),.rtaddrEX(rtaddrEX),.rtaddrID(rtaddrID),.rsaddrID(rsaddrID
),.PCSrcID(PCSrcID),
        .MemReadMEM(MemReadMEM),.MemtoRegMEM(MemtoRegMEM),.regwriteaddrMEM(regwriteaddrM
EM),//input data and sig
        .stall(stall),.flush(flush));//output sig

    ID EX ID EX(.clk(clk),.reset(reset),.intterupt(intterupt),
        .readdata1ID(readdata1ID),.readdata2ID(readdata2ID),.extenddataID(extenddataID),
.PCplus4ID(PCplus4ID),
```

```
       .rdaddrID(rdaddrID),.rtaddrID(rtaddrID),.rsaddrID(rsaddrID),.shamtID(shamtID),.P
CID(PCID),
       .RegWriteID(RegWriteID),.ExtOpID(ExtOpID),.MemReadID(MemReadID),
       .MemWriteID(MemWriteID),.ALUSrcID(ALUSrcID),.MemtoRegID(MemtoRegID),
       .RegDstID(RegDstID),.ALUOpID(ALUOpID),.FunctID(FunctID),
       .readdata1EX(readdata1EX),.readdata2EX(readdata2EX),.extenddataEX(extenddataEX),
.PCplus4EX(PCplus4EX),
       .rdaddrEX(rdaddrEX),.rtaddrEX(rtaddrEX),.rsaddrEX(rsaddrEX),.shamtEX(shamtEX),.P
CEX(PCEX),
       .RegWriteEX(RegWriteEX),.ExtOpEX(ExtOpEX),.MemReadEX(MemReadEX),
       .MemWriteEX(MemWriteEX),.ALUSrcEX(ALUSrcEX),.MemtoRegEX(MemtoRegEX),
       .RegDstEX(RegDstEX),.ALUOpEX(ALUOpEX),.FunctEX(FunctEX));

   //wire [31:0] branchaddrEX;
   wire [31:0] ALUoutEX;
   wire [31:0] memwritedataEX;
   //wire [4:0] regwriteaddrEX;
   wire ALUequalEX;
   //wire [31:0] ALUoutMEM;
   //wire [4:0] regwriteaddrMEM;
   //wire RegWriteMEM;//forward input

   EX EX(.clk(clk),.reset(reset),.intterupt(intterupt),
       .readdata1EX(readdata1EX),.readdata2EX(readdata2EX),.extenddataEX(extenddataEX),
       .rdaddrEX(rdaddrEX),.rtaddrEX(rtaddrEX),.rsaddrEX(rsaddrEX),.shamtEX(shamtEX),
       .regwritedataWB(regwritedataWB),.ALUoutMEM(ALUoutMEM),.regwriteaddrWB(regwritead
drWB),.regwriteaddrMEM(regwriteaddrMEM),//input data
       .RegWriteWB(RegWriteWB),.RegWriteMEM(RegWriteMEM),
       .ALUSrcEX(ALUSrcEX),.ALUOpEX(ALUOpEX),.RegDstEX(RegDstEX),.FunctEX(FunctEX),//in
put control signals
       .ALUequalEX(ALUequalEX),//output control signals
       .ALUoutEX(ALUoutEX),.memwritedataEX(memwritedataEX),.regwriteaddrEX(regwriteaddr
EX));//output data

   //wire [31:0] branchaddrMEM;//used in IF
   //wire [31:0] ALUoutMEM;//used in EX
   wire [31:0] memwritedataMEM;
   //wire [4:0] regwriteaddrMEM;//uesd in EX
   wire [31:0] PCplus4MEM,PCMEM;
   wire ALUequalMEM,MemWriteMEM;//MemReadMEM,RegWriteMEM;
   //wire [1:0] MemtoRegMEM;

   EX MEM EX MEM(.clk(clk),.reset(reset),.intterupt(intterupt),
       .ALUequalEX(ALUequalEX),.MemWriteEX(MemWriteEX),.MemReadEX(MemReadEX),.MemtoRegE
X(MemtoRegEX),.RegWriteEX(RegWriteEX),
       .ALUoutEX(ALUoutEX),.memwritedataEX(memwritedataEX),.regwriteaddrEX(regwriteaddr
EX),.PCplus4EX(PCplus4EX),.PCEX(PCEX),
       .ALUequalMEM(ALUequalMEM),.MemWriteMEM(MemWriteMEM),.MemReadMEM(MemReadMEM),.Mem
toRegMEM(MemtoRegMEM),.RegWriteMEM(RegWriteMEM),
       .ALUoutMEM(ALUoutMEM),.memwritedataMEM(memwritedataMEM),.regwriteaddrMEM(regwrit
eaddrMEM),.PCplus4MEM(PCplus4MEM),.PCMEM(PCMEM));

   wire [31:0] memreaddataMEM;
   //wire PCSrcMEM;//used in IF

   MEM MEM(.clk(clk),.reset(reset),.intterupt(intterupt),.leds(leds),.digi(digi),
       .memaddrMEM(ALUoutMEM),.memwritedataMEM(memwritedataMEM),.check(PCIF[31]),.showa
ddr(showaddr),//input data
       .ALUequalMEM(ALUequalMEM),.MemWriteMEM(MemWriteMEM),.MemReadMEM(MemReadMEM),//in
put control signals
       .memreaddataMEM(memreaddataMEM),.ramshowdata(ramshowdata));//output data and
control signals

   wire [31:0] ALUoutWB;
   wire [31:0] memreaddataWB;
   wire [31:0] PCplus4WB,PCWB;
   //wire [4:0] regwriteaddrWB;//used in ID
   wire [1:0] MemtoRegWB;
   //wire RegWriteWB;//used in ID

   MEM WB MEM WB(.clk(clk),.reset(reset),.intterupt(intterupt),
       .ALUoutMEM(ALUoutMEM),.memreaddataMEM(memreaddataMEM),.regwriteaddrMEM(regwritea
ddrMEM),
       .MemtoRegMEM(MemtoRegMEM),.RegWriteMEM(RegWriteMEM),.PCplus4MEM(PCplus4MEM),.PCM
EM(PCMEM),
```

```
        .ALUoutWB(ALUoutWB),.memreaddataWB(memreaddataWB),.regwriteaddrWB(regwriteaddrWB
),
        .MemtoRegWB(MemtoRegWB),.RegWriteWB(RegWriteWB),.PCplus4WB(PCplus4WB),.PCWB(PCWB
));

    //wire [31:0] regwritedataWB;//used in ID

    WB WB(.clk(clk),.reset(reset),.intterupt(intterupt),
        .PCWB(PCWB),.PCplus4WB(PCplus4WB),.memreaddataWB(memreaddataWB),.ALUoutWB(ALUout
WB),.MemtoRegWB(MemtoRegWB),//input
        .regwritedataWB(regwritedataWB));//output

endmodule
```

上述代码就囊括了包括外设输出在内的整个 CPU 结构。每个模块具体的实现就不一一贴入报告了，可见附件的各个模块对应的.v 文件。值得注意的是数据转发与阻塞逻辑产生模块，代码如下：

```
module DataForward(clk,reset,intterupt,
    readdata1,readdata2,
    regwritedataWB,ALUoutMEM,
    regwriteaddrWB,regwriteaddrMEM,rsaddr,rtaddr,
    RegWriteWB,RegWriteMEM,
    forwardout1,forwardout2);

    input clk,reset,intterupt;
    input [31:0] readdata1,readdata2,regwritedataWB,ALUoutMEM;
    input [4:0] regwriteaddrWB,regwriteaddrMEM,rsaddr,rtaddr;
    input RegWriteWB,RegWriteMEM;
    output [31:0] forwardout1,forwardout2;

    wire [1:0] Forward1,Forward2;//00 choose readdata,01 choose ALUoutMEM,10 choose
regwritedataWB
    assign Forward1=(RegWriteMEM & (regwriteaddrMEM!=0) & (regwriteaddrMEM==rsaddr))?
2'b01:
        (RegWriteWB & (regwriteaddrWB!=0) & (regwriteaddrWB==rsaddr))? 2'b10:2'b00;
    assign Forward2=(RegWriteMEM & (regwriteaddrMEM!=0) & (regwriteaddrMEM==rtaddr))?
2'b01:
        (RegWriteWB & (regwriteaddrWB!=0) & (regwriteaddrWB==rtaddr))? 2'b10:2'b00;

    assign forwardout1=(Forward1==2'b01)? ALUoutMEM:(Forward1==2'b10)?
regwritedataWB:readdata1;
    assign forwardout2=(Forward2==2'b01)? ALUoutMEM:(Forward2==2'b10)?
regwritedataWB:readdata2;
endmodule


module StallFlush(clk,reset,intterupt,
    MemReadEX,rtaddrEX,rtaddrID,rsaddrID,PCSrcID,BranchID,JRID,regwriteaddrEX,
    MemReadMEM,regwriteaddrMEM,MemtoRegMEM,RegWriteEX,
    ,stall,flush);

    input clk,reset,intterupt;
    input MemReadEX,BranchID,JRID,MemReadMEM,RegWriteEX;
    input [4:0] regwriteaddrEX,rtaddrEX,rsaddrID,rtaddrID,regwriteaddrMEM;
    input [2:0] PCSrcID;
    input [1:0] MemtoRegMEM;
    output stall,flush;

    assign stall=(MemReadEX & (rtaddrEX!=0) & ((rtaddrEX==rsaddrID) |
(rtaddrEX==rtaddrID)))//load-use
    | (BranchID & RegWriteEX & (regwriteaddrEX!=0) & (regwriteaddrEX==rsaddrID |
regwriteaddrEX==rtaddrID))//R and beq
    | (BranchID & MemReadMEM & (regwriteaddrMEM!=0) & ((regwriteaddrMEM==rsaddrID) |
(regwriteaddrMEM==rtaddrID)))//lw and beq
    | (BranchID & (MemtoRegMEM==2'b10) & (rsaddrID==5'b11111 |
rtaddrID==5'b11111))//jal and beq
    | (BranchID & (MemtoRegMEM==2'b11) & (rsaddrID==5'b11010 |
rtaddrID==5'b11010))//$26 and beq
    | (JRID & RegWriteEX & (regwriteaddrEX!=0) & (regwriteaddrEX==rsaddrID))//R and jr
    | (JRID & MemReadMEM & (regwriteaddrMEM!=0) & (regwriteaddrMEM==rsaddrID))//lw and
jr
    | (JRID & (MemtoRegMEM==2'b10) & rsaddrID==5'b11111)//jal and jr
    | (JRID & (MemtoRegMEM==2'b11) & rsaddrID==5'b11010);//$26 and jr

    assign flush=(stall!=1) & (PCSrcID!=0);
endmodule
```

另外，最终 CPU 运行的程序，包括排序与中断处理如下：

```
    j main
    j interrupt
    j exception

main:
    lui $a0,0x4000#$a0 is the begin of peripherals
    addi $a1,$0,0#$a1 loop 0~3 to control AN
    addi $a2,$0,0#$a2 is the data to show on digi
    addi $a3,$0,0#$a3 is the decode result writing into 0x40000010
    addi $t4,$0,1
    addi $t5,$0,2
    addi $t6,$0,3#parameter 1,2,3
    addi $t7,$0,-7#$t7=0xfffffff9
    addi $t8,$0,-1#$t8=0xffffffff

sort:
    lw $a2,20($a0)#save start time
    addi $s1,$0,0#$s1 is the begin of datamemory
    addi $s2,$0,99#s2=N-1
    addi $s3,$0,1
loop1:
    bgt $s3,$s2,exit1
    addi $s4,$s3,-1
loop2:
    blt $s4,$0,exit2
    sll $t0,$s4,2
    add $t0,$s1,$t0
    lw $t1,0($t0)
    lw $t2,4($t0)
    blt $t2,$t1,swap
conti:
    addi $s4,$s4,-1
    j loop2
exit2:
    addi $s3,$s3,1
    j loop1
exit1:
    lw $t0,20($a0)
    sub $a2,$t0,$a2#cal a2
    srl $t0,$a2,16
    sw $t0,12($a0)#high in led
    addi $s0,$0,-60000
    sw $s0,0($a0)#set TH
    sw $t8,4($a0)#set TL
    sw $t6,8($a0)#open timer
    j end
swap:
    sll $t1,$s4,2
    add $t1,$s1,$t1
    lw $t0,0($t1)
    lw $t2,4($t1)
    sw $t2,0($t1)
    sw $t0,4($t1)
    j conti

interrupt:
    lw $t0,8($a0)
    and $t0,$t0,$t7
    sw $t0,8($a0)#TCON &= 0xfffffff9
    #save datas at $sp
    beq $a1,$0,show0
    beq $a1,$t4,show1
    beq $a1,$t5,show2
    beq $a1,$t6,show3

return:
    sw $a3,16($a0)
    bne $a1,$t6,a1plus
    addi $a1,$0,-1
a1plus:
    addi $a1,$a1,1
    #take datas from $sp
    lw $t0,8($a0)
    ori $t0,$t0,2
    sw $t0,8($a0)
```

```
    jr $26

show0:
    andi $s5,$a2,0x000f
    jal decode
    addi $a3,$a3,0x0100
    j return

show1:
    andi $s5,$a2,0x00f0
    srl $s5,$s5,4
    jal decode
    addi $a3,$a3,0x0200
    j return

show2:
    andi $s5,$a2,0x0f00
    srl $s5,$s5,8
    jal decode
    addi $a3,$a3,0x0400
    j return

show3:
    andi $s5,$a2,0xf000
    srl $s5,$s5,16
    jal decode
    addi $a3,$a3,0x0800
    j return

decode:
    addi $a3,$0,0
    addi $t0,$s5,0
    beq $t0,$0,zero
    addi $t0,$s5,-1
    beq $t0,$0,one
    addi $t0,$s5,-2
    beq $t0,$0,two
    addi $t0,$s5,-3
    beq $t0,$0,three
    addi $t0,$s5,-4
    beq $t0,$0,four
    addi $t0,$s5,-5
    beq $t0,$0,five
    addi $t0,$s5,-6
    beq $t0,$0,six
    addi $t0,$s5,-7
    beq $t0,$0,seven
    addi $t0,$s5,-8
    beq $t0,$0,eight
    addi $t0,$s5,-9
    beq $t0,$0,nine
    addi $t0,$s5,-10
    beq $t0,$0,ten
    addi $t0,$s5,-11
    beq $t0,$0,eleven
    addi $t0,$s5,-12
    beq $t0,$0,twelve
    addi $t0,$s5,-13
    beq $t0,$0,thirteen
    addi $t0,$s5,-14
    beq $t0,$0,fourteen
    addi $t0,$s5,-15
    beq $t0,$0,fifteen

zero:
    addi $a3,$0,0x003f
    jr $ra
one:
    addi $a3,$0,0x0006
    jr $ra
two:
    addi $a3,$0,0x005b
    jr $ra
three:
    addi $a3,$0,0x004f
    jr $ra
```

```
four:
    addi $a3,$0,0x0066
    jr $ra
five:
    addi $a3,$0,0x006d
    jr $ra
six:
    addi $a3,$0,0x007d
    jr $ra
seven:
    addi $a3,$0,0x0007
    jr $ra
eight:
    addi $a3,$0,0x007f
    jr $ra
nine:
    addi $a3,$0,0x006f
    jr $ra
ten:
    addi $a3,$0,0x0077
    jr $ra
eleven:
    addi $a3,$0,0x007c
    jr $ra
twelve:
    addi $a3,$0,0x0039
    jr $ra
thirteen:
    addi $a3,$0,0x005e
    jr $ra
fourteen:
    addi $a3,$0,0x0079
    jr $ra
fifteen:
    addi $a3,$0,0x0071
    jr $ra

exception:
    jr $26

end:
    j end
```

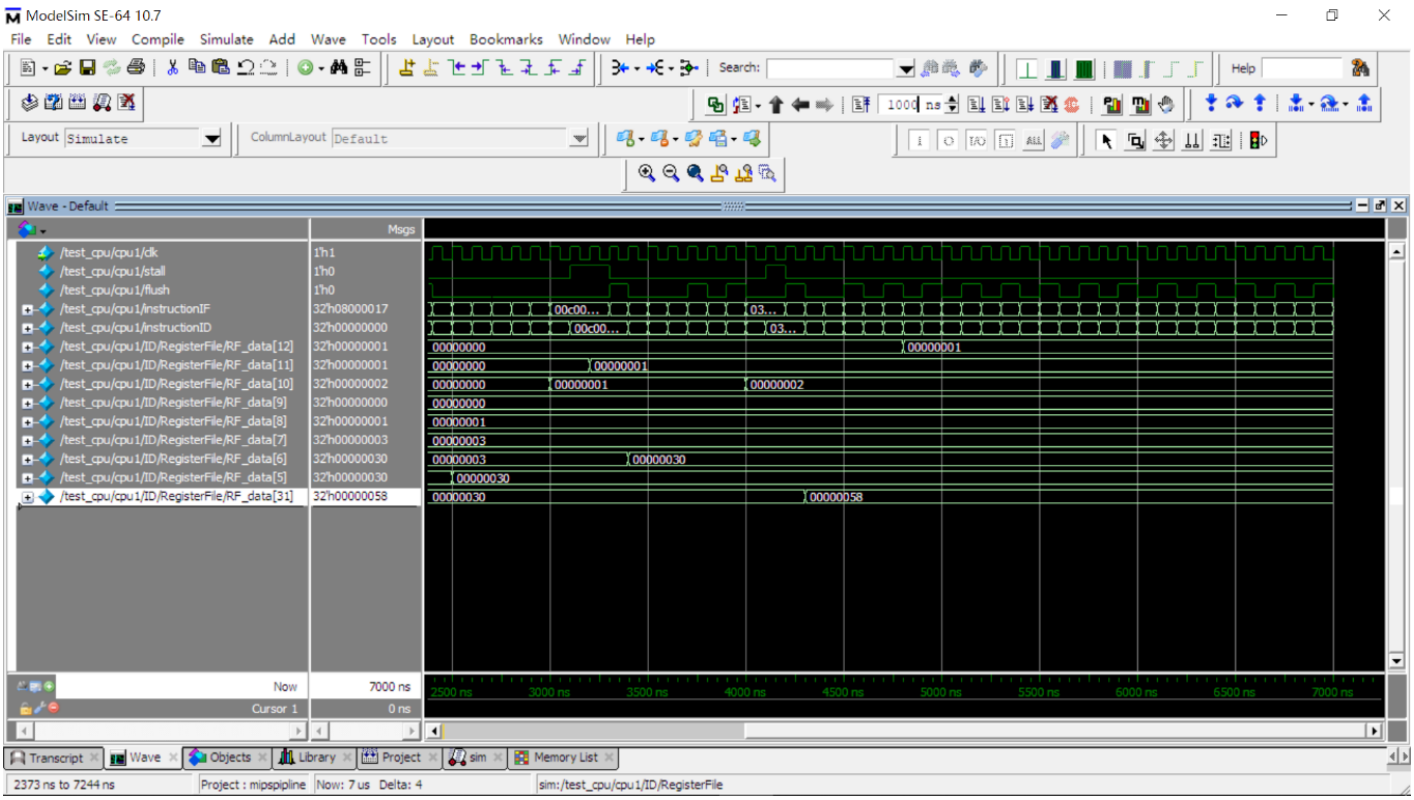
上述程序关于中断的打开、关闭以及一些寄存器的作用都可见注释。由于我们是在排序完成后才打开中断，因此也并不需要“保存现场”。实际的中断处理程序应该将所有寄存器值存入栈中，在处理结束时恢复这些值。

四、 仿真验证

算术、逻辑以及储存访问指令的仿真验证较为容易，这里我们直接进行分支与跳转的验证。同时这也囊括了大量的转发与阻塞逻辑。我们设计了如下程序，其包含了 stallflush 中涉及的所有阻塞情形。

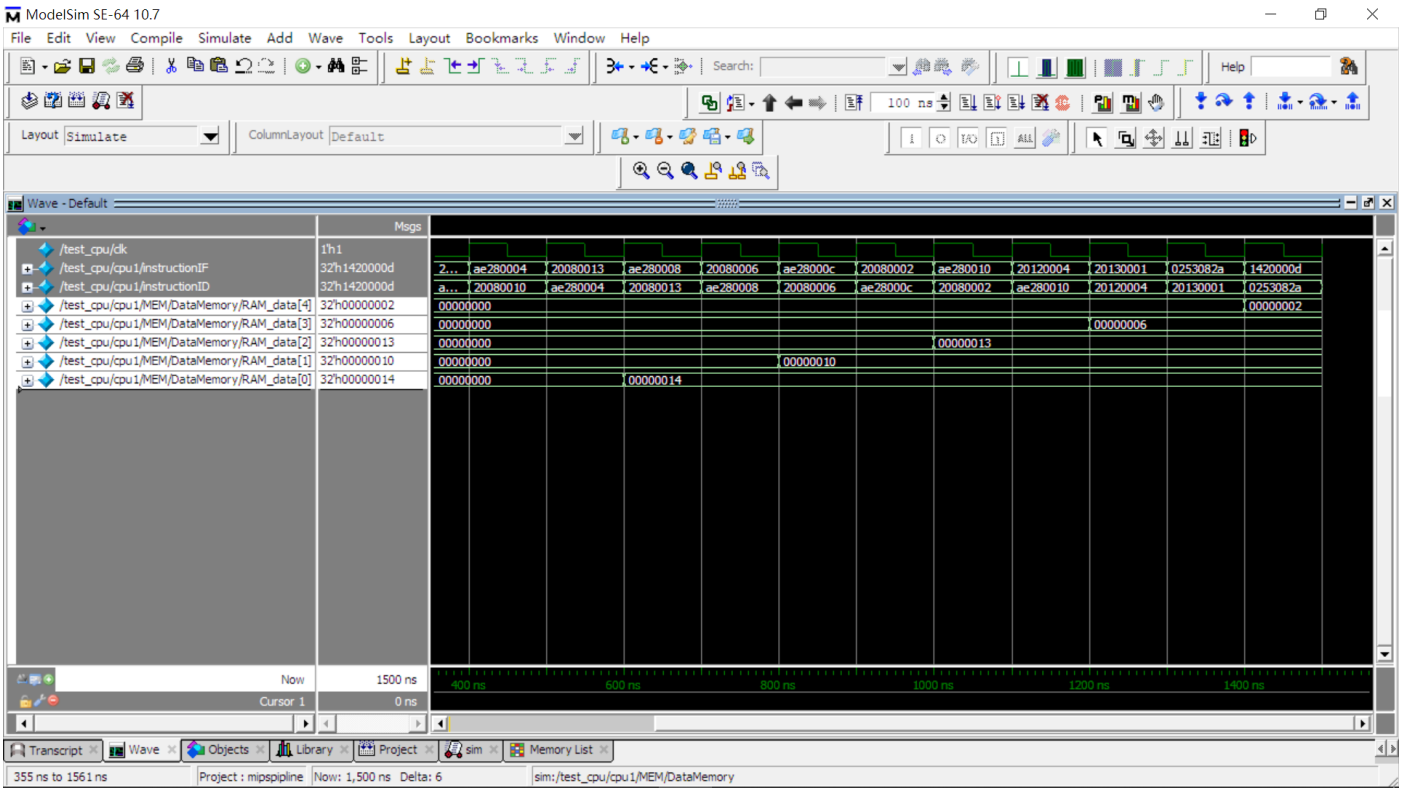
```
.data
    in buff: .space 4096
.text
    la $a0,in buff
    addi $a1,$0,3
    sw $a1,0($a0)
    lw $a2,0($a0)
    addi $a3,$a2,1#load-use
    beq $a3,$a2,branch1#R and beq
    addi $t0,$t0,1
branch1:
    lw $a3,0($a0)
    beq $a3,$a2,branch2#lw and beq
    addi $t1,$t1,1
branch2:
    jal jump1
jump1:
    bne $a1,$ra,branch3#jal and beq
    addi $t2,$t2,1
    beq $a1,$a2,branch4
    addi $t3,$t3,1
    sw $a1,0($a0)
    lw $a2,0($a0)
    jr $a2#lw and jr
branch3:
    addi $a1,$ra,0
    jr $a1#R and jr
branch4:
    jal jump2
    addi $t4,$t4,1
loop:
    j loop
jump2:
    jr $ra#jal and jr
```

为了便于观察程序运行结果，我们令$t0-$t4 作为计数器统计程序是否正确地执行了各个分支。我们设计的流水线 MIPS 处理器得到的结果与 MARS 仿真结果一致如下图:

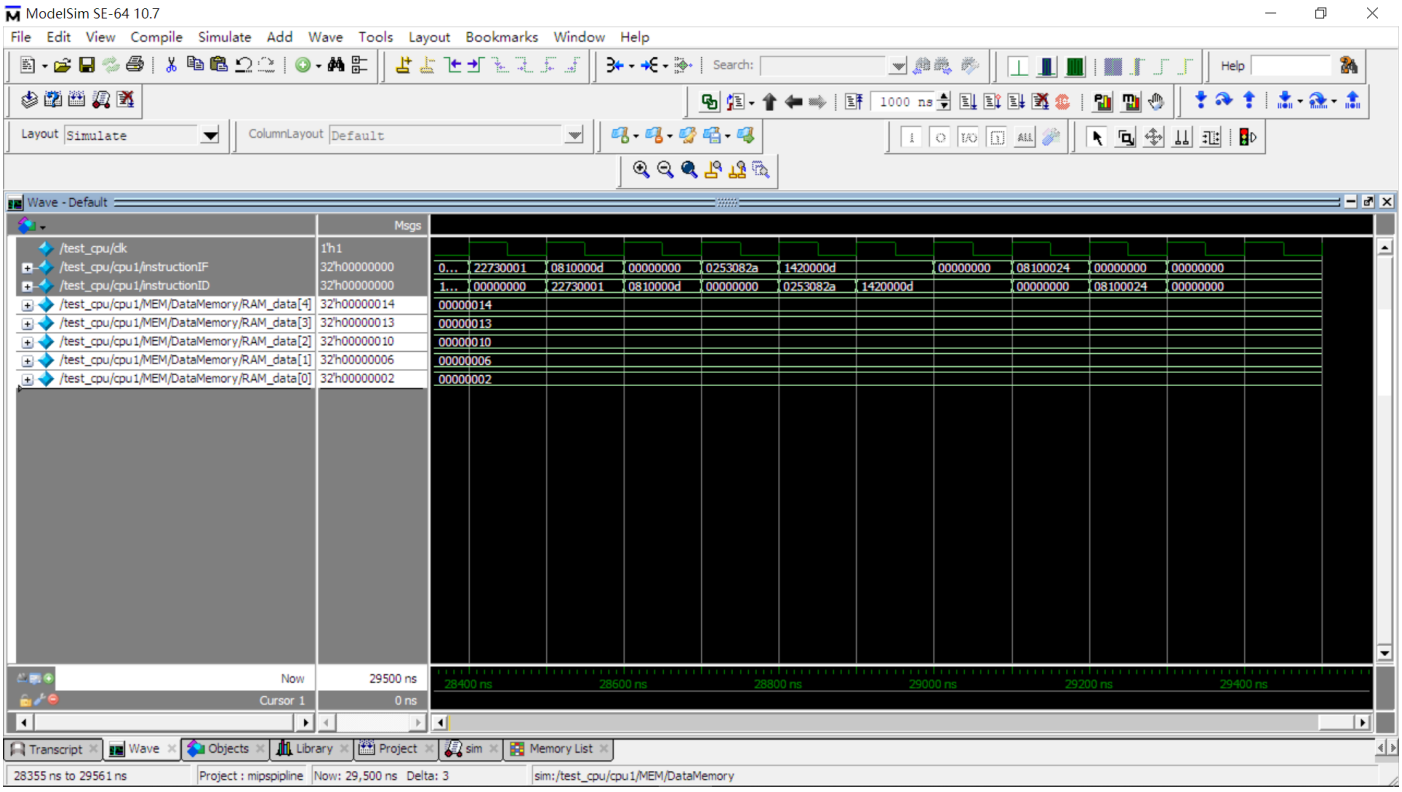


$a1	5	0x00400030
$a2	6	0x00400030
$a3	7	0x00000003
$t0	8	0x00000001
$t1	9	0x00000000
$t2	10	0x00000002
$t3	11	0x00000001
$t4	12	0x00000001

至此，我们有信心让我们的处理器完成一段复杂的、包含多个分支与跳转的排序程序。我们采用 lw 指令直接写入随机数以便于仿真。在仿真中我们展示 5 个数的排序如下图：

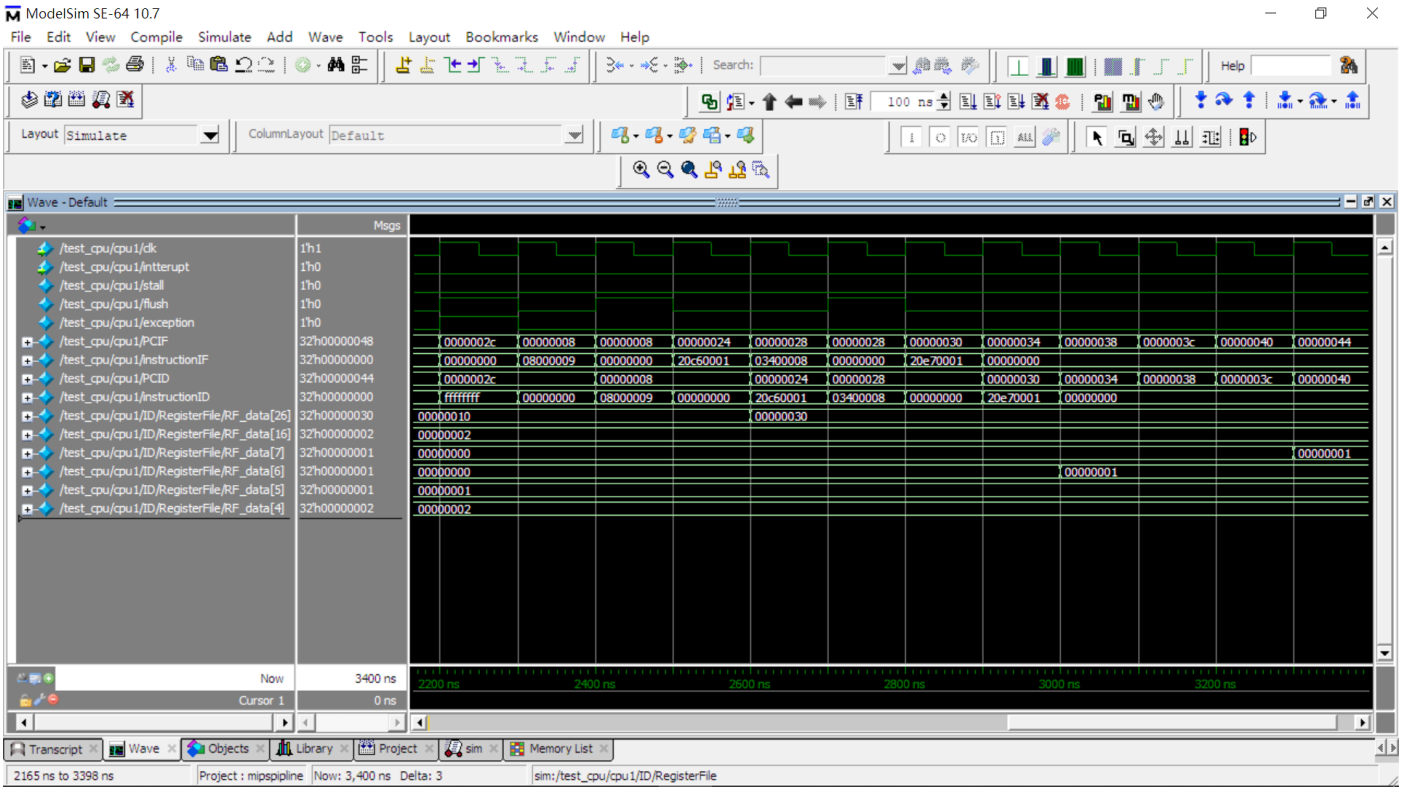


5 个数据写到 RAM 中



经过约 280 个周期排序完成

最后在仿真中验证异常处理功能。我们加入一条内容为 ffffffff 的指令，并观察 26 号寄存器。可以看到，该异常指令的下一条指令的地址被存入$26，并通过异常处理程序跳回到这里使程序继续运行。



未知指令异常及最终结果

有了这些仿真结果，我们再将外设对应的存储空间与 led、数码管等绑定，就完成了流水线处理器的设计。

五、 综合情况

我们查看 Vivado 的时序报告如下图:

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-1.609 ns	Worst Hold Slack (WHS):	0.071 ns	Worst Pulse Width Slack (WPWS):	4.146 ns
Total Negative Slack (TNS):	-86.862 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	107	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	11052	Total Number of Endpoints:	11052	Total Number of Endpoints:	5814

Timing constraints are not met.

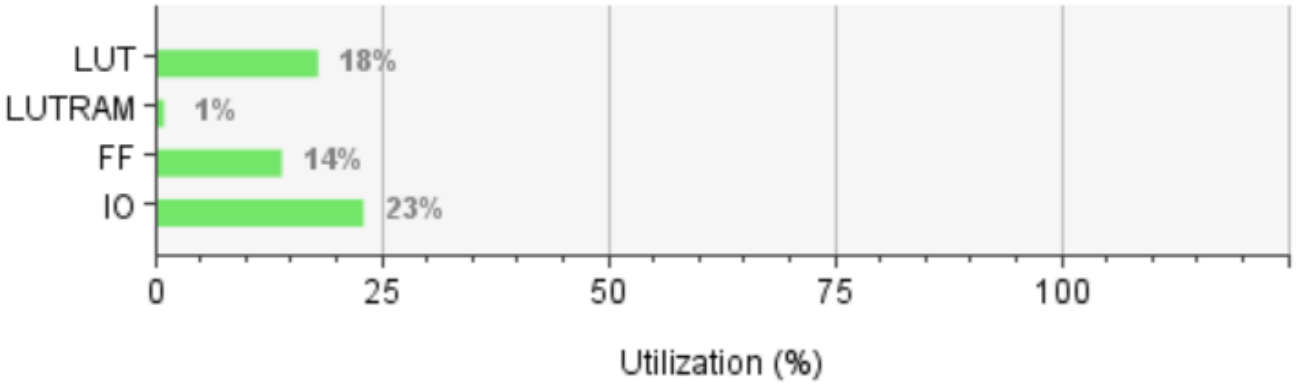
Intra-Clock Paths - CLK - Setup

Name	Slack	Levels	High Fanout	From	To	Total Delay
Path 1	-1.609	26	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/ID_EX/PCE...PCEX_reg_c/D	11.352
Path 2	-1.563	26	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/ID_EX/PCE...PCEX_reg_c/D	11.301
Path 3	-1.199	26	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/IF/PC_reg[2]/D	11.141
Path 4	-1.169	25	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/IF/PC_reg[27]/CE	10.883
Path 5	-1.167	25	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/IF/PC_reg[11]/CE	10.848
Path 6	-1.167	25	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/IF/PC_reg[15]/CE	10.848
Path 7	-1.167	25	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/IF/PC_reg[22]/CE	10.848
Path 8	-1.145	25	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/IF/PC_reg[5]/CE	10.888
Path 9	-1.125	26	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/IF/PC_reg[1]/D	11.070
Path 10	-1.104	25	108	CPU/ID/Register...ta_reg[23][9]/C	CPU/IF/PC_reg[25]/CE	10.815

可以看见，我们的设计并不能满足 100MHz 的时序要求。从总延时来看，我们的处理器在 90MHz 左右工作是没有问题的。不过我们将 FPGA100MHz 的系统时钟直接接入 clk，排序程序和中断都能够照常进行。观察延时最长的路径，集中在 ID 阶段，我们也许可以将 ID 阶段的一些计算放到 EX 阶段去完成，来加快整个处理器的速度。

资源占用报告如下图：

Resource	Utilization	Available	Utilization %
LUT	3680	20800	17.69
LUTRAM	16	9600	0.17
FF	5794	41600	13.93
IO	49	210	23.33



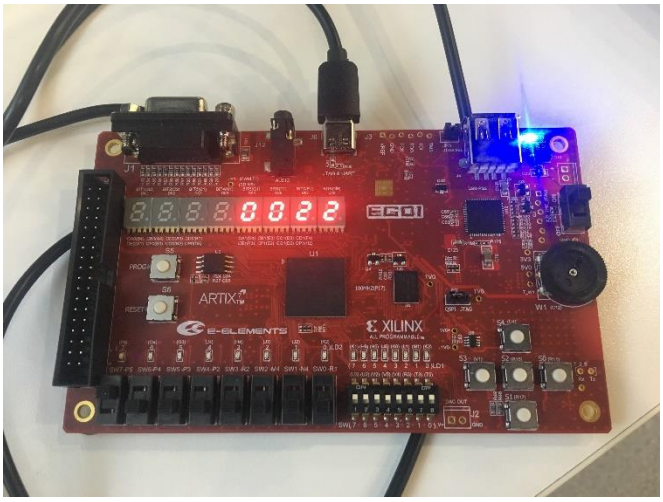
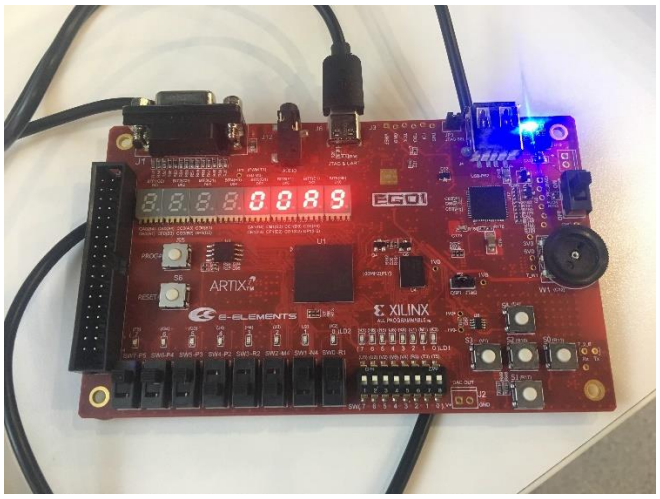
Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)	Bonded IOB (210)	BUFGCTRL (32)
top	3680	5794	1065	492	2697	3664	16	283	49	1
CPU (CPU)	3667	5764	1065	492	2683	3651	16	275	0	0
ramshow (...	13	30	0	0	17	13	0	5	0	0

共占用了 FPGA20%左右的资源。通过减少一些不用的端口，也许可以加快运行速度、减少资源占用。

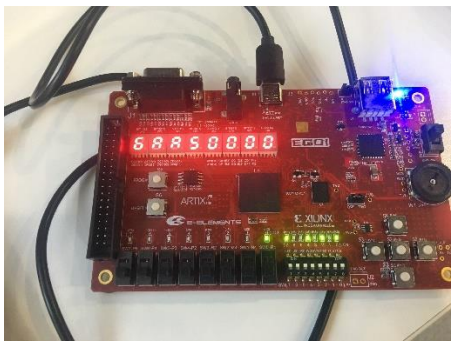
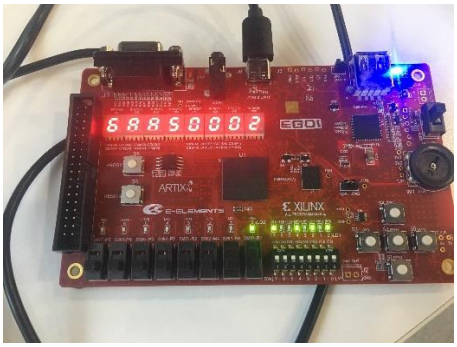
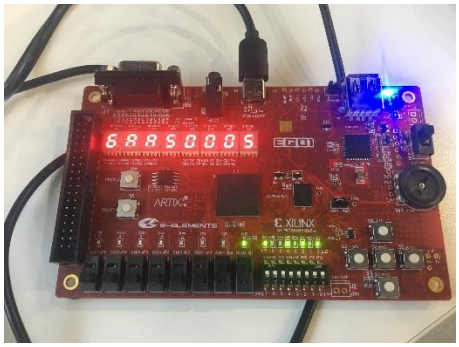
六、 硬件调试

我们通过硬件方式完成数据的输入与输出：在 reset 时，向 DataMemory 中写入待排序的 100 个数据。然后以 7 个开关作为地址来查看这 100 个位置的值。由于仿真已经做了充分的 debug，我们在硬件上没有碰到太大的问题。一个问题是一开始将 interrupt 这个变量的类型在 MEM 中错误设计为 input，导致其无法正常输出。在我们的处理器结构中，interrupt 是由 MEM 中的外设 output 的。我们通过将各个变量的值显示在 led 上的方法完成了 debug，过程相当顺利。

程序最终运行的情况如下图：

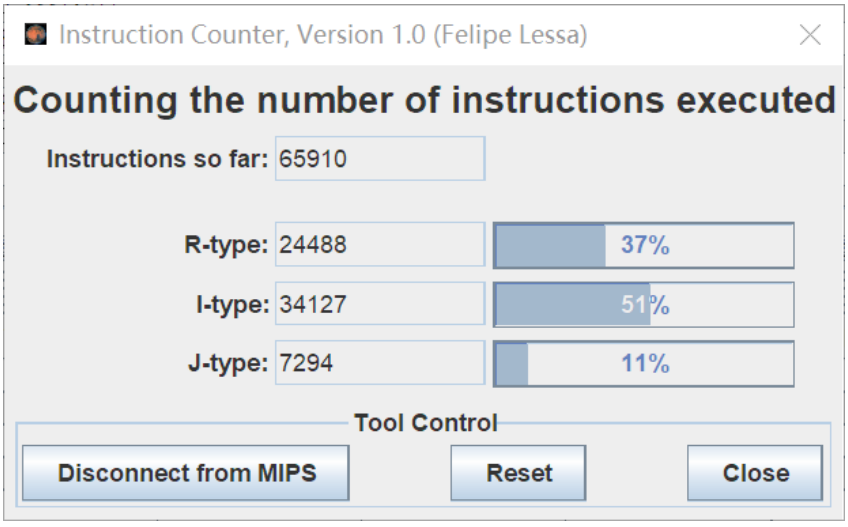
reset 状态下内存中第 0 与 99 个数

程序运行结束后内存中第 0,1,3 个数，以及系统时钟经过的周期数(最高位在 led 上)

七、 总结

综上，我们所设计的流水线 MIPS 处理器成功的工作了。我们将随机数排序程序用 MARS 的仿真功能得到指令数为 65910 条，而最终得到的时钟周期数为 0x16aa5，即 92837 个时钟周期。我们处理器的 CPI 约为 1.4。



CPI 主要受到数据与控制冒险的影响。我们的程序也的确有多处 J 型指令、跳转指令和 load-use 型阻塞。

通过本次实验，我们亲手实现了春季学期学习过的流水线结构，对其有了更深刻的理解，尤其是在转发、阻塞与控制冒险等方面有了新的认识。尽管过程相当艰辛，但最终能够看到数据顺利的出现在仿真界面和数码管上时，内心的喜悦与成就感十足。在编写处理器的过程中，我认为最重要的是两点，一是程序编写要有条理，由于处理器结构复杂，有十几个模块，一些模块的参数多达二三十个，此时变量的命名以及注释就非常重要；二是 debug 的手段，主要是如何使用仿真软件。在仿真时要明确自己这段程序的目的，哪些变量是关键变量，哪些变量对 debug 有帮助(如 PC，instruction，stall，flush 这样的变量应当时刻查看)，不能因为程序没跑通就瞎 de 一气。理清逻辑，一个个变量进行逻辑上的回溯，debug 就又快又好了。

本次实验的 CPU 部分大约花费了 30 个小时的时间，而外设部分又花费了不少时间。一开始我的确没有明白定时器中断的作用，最后也是向同学询问才知道相当于用中断代替我们之前用到的扫描信号，而外设作为存储的一部分，用 sw 指令控制数码管的显示。

最后，我们的实验也存在许多不足。异常处理部分非常简易，仅仅处理一条全 1 指令作为示例；时序上还有提升空间；一些废弃的端口没有进行删除，如 ALU 的 zero 部分；缺少 PC 和 PC+4 的转发，使得一些指令的 stall 时间过长等等。总体而言，这次实验让我对数字电路，对 Verilog，对 MIPS 指令和流水线结构都有了很大的提升。最后，感谢老师与助教几个月来的悉心指导以及验收时的专业认真！